Programming With the Intel MMX™ Technology 32

The Intel MMX technology comprises a set of extensions to the Intel Architecture that are designed to greatly enhance the performance of advanced media and communications applications. These extensions (which include new registers, data types, and instructions) are combined with a single-instruction, multiple-data (SIMD) execution model to accelerate the performance of applications such as motion video, combined graphics with video, image processing, audio synthesis, speech synthesis and compression, telephony, video conferencing, and 2D and 3D graphics, which typically use compute-intensive algorithms to perform repetitive operations on large arrays of simple, native data elements.

The MMX technology defines a simple and flexible software model, with no new mode or operating-system visible state. All existing software will continue to run correctly, without modification, on Intel Architecture processors that incorporate the MMX technology, even in the presence of existing and new applications that incorporate this technology.

The following sections of this chapter describe the MMX technology's basic programming environment, including the MMX register set, data types, and instruction set. Detailed descriptions of the MMX instructions are provided in Chapter 3, *Instruction Set Reference*, of the *Intel Architecture Software Developer's Manual, Volume 2*. The manner in which the MMX technology is integrated into the Intel Architecture system programming model is described in Chapter 10, *MMX™ Technology System Programming Model*, in the *Intel Architecture Software Developer's Manual, Volume 3*.

32.1 Overview of the MMX™ Technology Programming Environment

MMX technology provides the following new extensions to the Intel Architecture programming environment.

- Eight MMX™ registers (MM0 through MM7).
- Four MMX data types (packed bytes, packed words, packed doublewords, and quadword).
- The MMX instruction set.

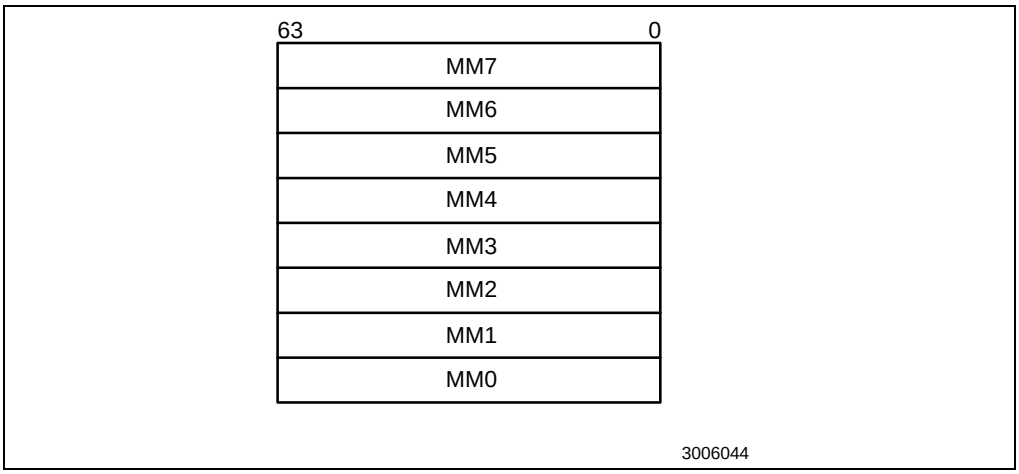
The MMX registers and data types are described in the following sections. See "Overview of the MMX™ Instruction Set", for an overview of the MMX instructions.

32.1.1 MMX™ Registers

The MMX register set consists of eight 64-bit registers (see Figure 32-1). The MMX instructions access the MMX registers directly using the register names MM0 through MM7. These registers can only be used to perform calculations on MMX data types; they cannot be used to address

memory. Addressing of MMX instruction operands in memory is handled by using the standard Intel Architecture addressing modes and general-purpose registers (EAX, EBX, ECX, EDX, EBP, ESI, EDI, and ESP).

Figure 32-1. MMX™ Register Set



Although the MMX registers are defined in the Intel Architecture as separate registers, they are aliased to the registers in the FPU data register stack (R0 through R7). (See Chapter 10, *MMX™ Technology System Programming Model*, in the *Intel Architecture Software Developer's Manual, Volume 3*, for more a detailed discussion of the aliasing of MMX registers.)

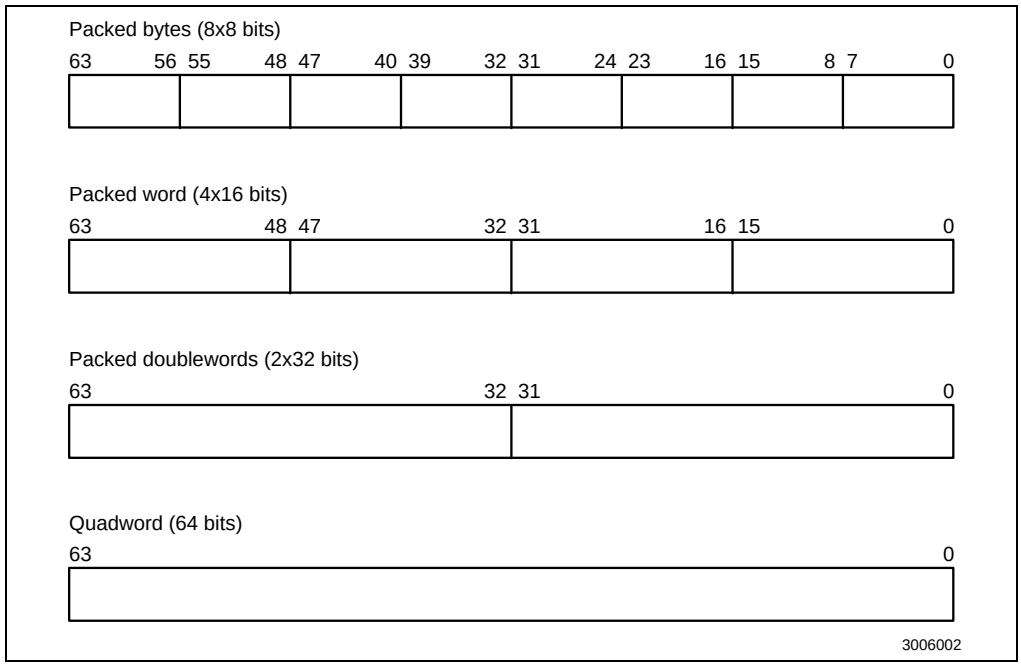
32.1.2 MMX™ Data Types

The MMX technology defines the following new 64-bit data types (see Figure 32-2):

Packed bytes	Eight bytes packed into one 64-bit quantity.
Packed words	Four (16-bit) words packed into one 64-bit quantity.
Packed doublewords	Two (32-bit) doublewords packed into one 64-bit quantity.
Quadword	One 64-bit quantity.

The bytes in the packed bytes data type are numbered 0 through 7, with byte 0 being contained in the least significant bits of the data type (bits 0 through 7) and byte 7 being contained in the most significant bits (bits 56 through 63). The words in the packed words data type are numbered 0 through 4, with word 0 being contained in the bits 0 through 15 of the data type and word 4 being contained in bits 48 through 63. The doublewords in a packed doublewords data type are numbered 0 and 1, with doubleword 0 being contained in bits 0 through 31 and doubleword 1 being contained in bits 32 through 63.

Figure 32-2. MMX™ Data Types



The MMX instructions move the packed data types (packed bytes, packed words, or packed doublewords) and the quadword data type to-and-from memory or to-and-from the Intel Architecture general-purpose registers in 64-bit blocks. However, when performing arithmetic or logical operations on the packed data types, the MMX instructions operate in parallel on the individual bytes, words, or doublewords contained in a 64-bit MMX register, as described in the following section (“Single Instruction, Multiple Data (SIMD) Execution Model”).

When operating on the bytes, words, and doublewords within packed data types, the MMX instructions recognize and operate on both signed and unsigned byte integers, word integers, and doubleword integers.

32.1.3 Single Instruction, Multiple Data (SIMD) Execution Model

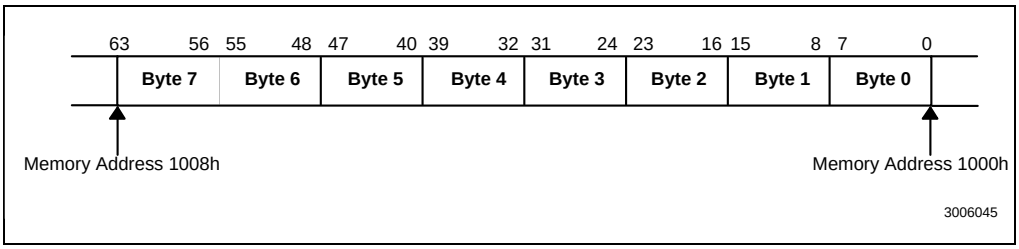
The MMX technology uses the single instruction, multiple data (SIMD) technique for performing arithmetic and logical operations on the bytes, words, or doublewords packed into 64-bit MMX registers. For example, the PADDSB instruction adds 8 signed bytes from the source operand to 8 signed bytes in the destination operand and stores 8 byte-results in the destination operand. This SIMD technique speeds up software performance by allowing the same operation to be carried out on multiple data elements in parallel. The MMX technology supports parallel operations on byte, word, and doubleword data elements when contained in MMX registers.

The SIMD execution model supported in the MMX technology directly addresses the needs of modern media, communications, and graphics applications, which often use sophisticated algorithms that perform the same operations on a large number of small data types (bytes, words, and doublewords). For example, most audio data is represented in 16-bit (word) quantities. The MMX instructions can operate on 4 of these words simultaneously with one instruction. Video and graphics information is commonly represented as palletized 8-bit (byte) quantities. Here, one MMX instruction can operate on 8 of these bytes simultaneously.

32.1.4 Memory Data Formats

When stored in memory the bytes, words, and doublewords in the packed data types are stored in consecutive addresses, with the least significant byte, word, or doubleword being stored in the at the lowest address and the more significant bytes, words, or doubleword being stored at consecutively higher addresses (see Figure 32-3). The ordering bytes, words, or doublewords in memory is always little endian. That is, the bytes with the lower addresses are less significant than the bytes with the higher addresses.

Figure 32-3. Eight Packed Bytes in Memory (at address 1000H)



32.1.5 Data Formats for MMX™ Registers

Values in MMX registers have the same format as a 64-bit quantity in memory. MMX registers have two data access modes: 64-bit access mode and 32-bit access mode.

The 64-bit access mode is used for 64-bit memory access, 64-bit transfer between MMX registers, all pack, logical and arithmetic instructions, and some unpack instructions.

The 32-bit access mode is used for 32-bit memory access, 32-bit transfer between integer registers and MMX registers, and some unpack instructions.

32.2 MMX™ Instruction Set

The MMX instruction set consists of 57 instructions, grouped into the following categories:

- Data transfer instructions
- Arithmetic instructions
- Comparison instructions
- Conversion instructions
- Logical instructions
- Shift instructions
- Empty MMX™ state instruction (EMMS)

When operating on packed data within an MMX register, the data is cast by the type specified by the instruction. For example, the PADDB (add packed bytes) instruction treats the packed data in an MMX register as 8 packed bytes; whereas, the PADDW (add packed words) instruction treats the packed data as 4 packed words.

32.2.1 Saturation Arithmetic and Wraparound Mode

The MMX technology supports a new arithmetic capability known as saturating arithmetic. Saturation is best defined by contrasting it with wraparound mode.

In wraparound mode, results that overflow or underflow are truncated and only the lower (least significant) bits of the result are returned; that is, the carry is ignored.

In saturation mode, results of an operation that overflow or underflow are clipped (saturated) to a data-range limit for the data type (see Table 32-1). The result of an operation that exceeds the range of a data-type saturates to the maximum value of the range. A result that is less than the range of a data type saturates to the minimum value of the range. This method of handling overflow and underflow is useful in many applications, such as color calculations.

Table 32-1. Data Range Limits for Saturation

Data Type	Lower Limit		Upper Limit	
	Hexadecimal	Decimal	Hexadecimal	Decimal
Signed Byte	80H	-128	7FH	127
Signed Word	8000H	-32,768	7FFFH	32,767
Unsigned Byte	00H	0	FFH	255
Unsigned Word	0000H	0	FFFFH	65,535

For example, when the result exceeds the data range limit for signed bytes, it is saturated to 7FH (FFH for unsigned bytes). If a value is less than the data range limit, it is saturated to 80H for signed bytes (00H for unsigned bytes).

Saturation provides a useful feature of avoiding wraparound artifacts. In the example of color calculations, saturation causes a color to remain pure black or pure white without allowing for and inversion.

MMX instructions do not indicate overflow or underflow occurrence by generating exceptions or setting flags.

32.2.2 Instruction Operands

All MMX instructions, except the EMMS instruction, reference and operate on two operands: the source and destination operands. The first operand is the destination and the second operand is the source. The destination operand may also be a second source operand for the operation. The instruction overwrites the destination operand with the result.

For example, a two-operand instruction would be decoded as:

DEST (first operand) ← DEST (first operand) OPERATION SRC (second operand)

The source operand for all the MMX instructions (except the data transfer instructions), can reside either in memory or in an MMX register. The destination operand resides in an MMX register.

For data transfer instructions, the source and destination operands can also be an integer register (for the MOVD instruction) or memory location (for both the MOVD and MOVQ instructions).

32.3 Overview of the MMX™ Instruction Set

Table 32-2 shows the instructions in the MMX instruction set. The following sections give a brief overview of each group of instructions in the MMX instruction set and the instructions within each group.

32.3.1 Data Transfer Instructions

The MOVD (Move 32 Bits) instruction transfers 32 bits of packed data from memory to MMX registers and visa versa, or from integer registers to MMX registers and visa versa.

The MOVQ (Move 64 Bits) instruction transfers 64-bits of packed data from memory to MMX registers and vise versa, or transfers data between MMX registers.

Table 32-2. MMX™ Instruction Set Summary (Sheet 1 of 2)

Category		Wraparound	Signed Saturation	Unsigned Saturation
Arithmetic	Addition	PADDB, PADDW, PADDD	PADDSB, PADDSW	PADDUSB, PADDUSW
	Subtraction	PSUBB, PSUBW, PSUBD	PSUBSB, PSUBSW	PSUBUSB, PSUBUSW
	Multiplication	PMULL, PMULH		
	Multiply and Add	PMADD		
Comparison	Compare for Equal	PCMPEQB, PCMPEQW, PCMPEQD		
	Compare for Greater Than	PCMPGTPB, PCMPGTPW, PCMPGTPD		
Conversion	Pack		PACKSSWB, PACKSSDW	PACKUSWB
	Unpack High	PUNPCKHBW, PUNPCKHWD, PUNPCKHDQ		
	Unpack Low	PUNPCKLBW, PUNPCKLWD, PUNPCKLDQ		
Category		**Packed**	**Full Quadword**	
Logical	And		PAND	
	And Not		PANDN	
	Or		POR	
	Exclusive OR		PXOR	
Shift	Shift Left Logical	PSLLW, PSLLD	PSLLQ	
	Shift Right Logical	PSRLW, PSRLD	PSRLQ	
	Shift Right Arithmetic	PSRAW, PSRAD		

Table 32-2. MMX™ Instruction Set Summary (Sheet 2 of 2)

Category		Doubleword Transfers	Quadword Transfers
Data Transfer	Register to Register	MOVD	MOVQ
	Load from Memory	MOVD	MOVQ
	Store to Memory	MOVD	MOVQ
Empty MMX™ State		EMMS	

32.3.2 Arithmetic Instructions

The arithmetic instructions perform addition, subtraction, multiplication, and multiply/add operations on packed data types.

32.3.2.1 Packed Addition And Subtraction

The PADDSB, PADDSW, and PADDWD (packed add) and PSUBB, PSUBW, and PSUBD (packed subtract) instructions add or subtract the signed or unsigned data elements of the source operand to or from the destination operand in wrap- around mode. These instructions support packed byte, packed word, and packed doubleword data types.

The PADDSB and PADDSW (packed add with saturation) and PSUBSB and PSUBSW (packed subtract with saturation) instructions add or subtract the signed data elements of the source operand to or from the signed data elements of the destination operand and saturate the result to the limits of the signed data-type range. These instructions support packed byte and packed word data types.

The PADDUSB and PADDUSW (packed add unsigned with saturation) and PSUBUSB and PSUBUSW (packed subtract unsigned with saturation) instructions add or subtract the unsigned data elements of the source operand to or from the unsigned data elements of the destination operand and saturate the result to the limits of the unsigned data-type range. These instructions support packed byte and packed word data types.

32.3.2.2 Packed Multiplication

Packed multiplication instructions perform four multiplications on pairs of signed 16-bit operands, producing 32-bit intermediate results. Users may choose the low-order or high-order parts of each 32-bit result.

The PMULHW (packed multiply high) and PMULLW (packed multiply low) instructions multiply the signed words of the source and destination operands and write the high-order or low-order 16 bits of each of the results to the destination operand.

32.3.2.3 Packed Multiply Add

The PMADDWD (packed multiply and add) instruction calculates the products of the signed words of the source and destination operands. The four intermediate 32-bit doubleword products are summed in pairs to produce two 32-bit doubleword results.

32.3.3 Comparison Instructions

The PCMPEQB, PCMPEQW, and PCMPEQD (packed compare for equal) and PCMPGTB, PCMPGTW, and PCMPGTD (packed compare for greater than) instructions compare the corresponding data elements in the source and destination operands for equality or value greater than, respectively. These instructions generate a mask of ones or zeros which are written to the destination operand. Logical operations can use the mask to select elements. This can be used to implement a packed conditional move operation without a branch or a set of branch instructions. No flags are set.

These instructions support packed byte, packed word and packed doubleword data types.

32.3.4 Conversion Instructions

The conversion instructions convert the data elements within a packed data type.

The PACKSSWB and PACKSSDW (packed with signed saturation) instruction converts signed words into signed bytes or signed doublewords into signed words, in signed saturation mode.

The PACKUSWB (packed with unsigned saturation) instruction converts signed words into unsigned bytes, in unsigned saturation mode.

The PUNPCKHBW, PUNPCKHWD, and PUNPCKHDQ (unpack high packed data) and PUNPCKLBW, PUNPCKLWD, and PUNPCKLDQ (unpack low packed data) instructions convert bytes to words, words to doublewords, or doublewords to quadwords.

32.3.5 Logical Instructions

The PAND (bitwise logical AND), PANDN (bitwise logical AND NOT), POR (bitwise logical OR), and PXOR (bitwise logical exclusive OR) instructions perform bitwise logical operations on 64-bit quantities.

32.3.6 Shift Instructions

The logical shift left, logical shift right and arithmetic shift right instructions shift each element by a specified number of bits. The logical left and right shifts also enable a 64-bit quantity (quadword) to be shifted as one block, assisting in data type conversions and alignment operations.

The PSLLW and PSLLD (packed shift left logical) and PSRLW and PSRLD (packed shift right logical) instructions perform a logical left or right shift, and fill the empty high or low order bit positions with zeros. These instructions support packed word, packed doubleword, and quadword data types.

The PSRAW and PSRAD (packed shift right arithmetic) instruction performs an arithmetic right shift, copying the sign bit into empty bit positions on the upper end of the operand. This instruction supports packed word and packed doubleword data types.

32.3.7 EMMS (Empty MMX™ State) Instruction

The EMMS instruction empties the MMX state. This instruction must be used to clear the MMX state (empty the floating-point tag word) at the end of an MMX routine before calling other routines that can execute floating-point instructions.

32.4 Compatibility with FPU Architecture

The MMX state is aliased upon the Intel Architecture floating-point state. No new state or mode is added to support the MMX technology. The same floating-point instructions that save and restore the floating-point state also handle the MMX state (for example, during context switching).

MMX technology uses the same interface techniques between the floating-point architecture and the operating system (primarily for task switching purposes). For more details, see Chapter 10, *MMX™ Technology System Programming Model*, in the *Intel Architecture Software Developer's Manual, Volume 3*.

32.4.1 MMX™ Instructions and the Floating-Point Tag Word

After each MMX instruction, the entire floating-point tag word is set to Valid (00s). The Empty MMX state (EMMS) instruction sets the entire floating-point tag word to Empty (11s).

Chapter 10, *MMX™ Technology System Programming Model*, in the *Intel Architecture Software Developer's Manual, Volume 3*, describes the effects of floating-point and MMX instructions on the floating-point tag word. For details on floating-point tag word, see "FPU Tag Word".

32.4.2 Effect of Instruction Prefixes on MMX™ Instructions

Table 32-3 details the effect of an instruction prefix on an MMX instruction.

Table 32-3. Effect of Prefixes on MMX™ Instructions

Prefix Type	Effect of Prefix
Address size (67H)	Affects MMX™ instructions with a memory operand. Ignored by MMX instructions without a memory operand.
Operand size (66H)	Ignored.
Segment override	Affects MMX instructions with a memory operand. Ignored by MMX instructions without a memory operand.
Repeat	Ignored.
Lock (F0H)	Generates an invalid opcode exception.

See the section titled "Instruction Prefixes" in Chapter 2 of the *Intel Architecture Software Developer's Manual, Volume 2*, for detailed information on prefixes.

32.5 WRITING APPLICATIONS WITH MMX™ CODE

The following sections give guidelines for writing applications code uses the MMX technology.

32.5.1 Detecting Support for MMX™ Technology Using the CPUID Instruction

Use the CPUID instruction to determine whether the processor supports the MMX instruction set (see the section titled "CPUID—CPU Identification" in Chapter 3 of the *Intel Architecture Software Developer's Manual, Volume 2*, for a detailed description of the CPUID instruction). When the support for MMX technology is detected by the CPUID instruction, it is signaled by setting bit 23 (MMX technology bit) in the feature flags to 1. In general, two versions of the routine can be created: one with scalar instructions and one with MMX instructions. The application will call the appropriate routine depending on the results of the CPUID instruction. If support for MMX technology is detected, then the MMX routine is called; if no support for the MMX technology exists, the application calls the scalar routine.

Note: The CPUID instruction will continue to report the existence of the MMX technology if the CR0.EM bit is set (which signifies that the CPU is configured to generate exception interrupt 7 that can be used to emulate floating point instructions). In this case, executing an MMX instruction results in an invalid opcode exception.

Example 32-1 illustrates how to use the CPUID instruction. This example does not represent the entire CPUID sequence, but shows the portion used for detection of MMX technology.

Example 32-1. Partial Routine for Detecting MMX™ Technology with the CPUID Instruction

```
...     ; identify existence of CPUID instruction
...
...     ; identify Intel processor
....
mov EAX, 1; request for feature flags
CPUID   ; 0Fh, 0A2h CPUID instruction
testEDX, 00800000h; Is IA MMX technology bit (Bit 23 of EDX)
        ; in feature flags set?
jnz MMX_Technology_Found
```

32.5.2 Using the EMMS Instruction

When integrating an MMX routine into an application running under an existing operating system, programmers need to take special precautions, similar to those when writing floating-point code.

When an MMX instruction executes, the floating-point tag word is marked valid (00s). Subsequent floating-point instructions that will be executed may produce unexpected results because the floating-point stack seems to contain valid data. The EMMS instruction marks the floating-point tag word as empty. Therefore, it is imperative to use the EMMS instruction at the end of every MMX routine, if the next routine may contain FPU code.

The EMMS instruction must be used in each of the following cases:

- When an application using the floating-point instructions calls an MMX™ technology library/ DLL. (Use the EMMS instruction at the end of the MMX code.)

- When an application using MMX instructions calls a floating-point library/DLL. (Use the EMMS instruction before calling the floating-point code.)
- When a switch is made between MMX code in a task/thread and other tasks/threads in cooperative operating systems, unless it is certain that more MMX instructions will be executed before any FPU code.

If the EMMS instruction is not used when trying to execute a floating-point instruction, the following may occur:

- Depending on the exception mask bits of the floating-point control word, a floating- point exception event may be generated.
- A “soft exception” may occur. In this case floating-point code continues to execute, but generates incorrect results. This happens when the floating-point exceptions are masked and no visible exceptions occur. The internal exception handler (microcode, not user visible) loads a NaN (Not a Number) with an exponent of 11..11B onto the floating-point stack. The NaN is used for further calculations, yielding incorrect results.
- A potential error may occur only if the operating system does NOT manage floating-point context across task switches. These operating systems are usually cooperative operating systems. It is imperative that the EMMS instruction execute at the end of all the MMX™ routines that may enable a task switch immediately after they end execution (explicit yield API or implicit yield API).

32.5.3 Interfacing with MMX™ Code

The MMX technology enables direct access to all the MMX registers. This means that all existing interface conventions that apply to the use of the processor’s general-purpose registers (EAX, EBX, etc.) also apply to use of MMX register.

An efficient interface to MMX routines might pass parameters and return values through the MMX registers or through a combination of memory locations (via the stack) and MMX registers. Such an interface would have to be written in assembly language since passing parameters through MMX registers is not currently supported by any existing C compilers. Do not use the EMMS instruction when the interface to the MMX code has been defined to retain values in the MMX register.

If a high-level language, such as C, is used, the data types could be defined as a 64-bit structure with packed data types.

When implementing usage of MMX instructions in high-level languages other approaches can be taken, such as:

- Passing parameters to an MMX™ routine by passing a pointer to a structure via the integer stack.
- Returning a value from a function by returning the pointer to a structure.

32.5.4 Writing Code with MMX™ and Floating-Point Instructions

The MMX technology aliases the MMX registers on the floating-point registers. The main reason for this is to enable MMX technology to be fully compatible and transparent to existing software environments (operating systems and applications). This way operating systems will be able to include new applications and drivers that use the MMX technology.

An application can contain both floating-point and MMX code. However, the user is discouraged from causing frequent transitions between MMX and floating-point instructions by mixing MMX code and floating-point code.

32.5.4.1 RECOMMENDATIONS AND GUIDELINES

Do not mix MMX code and floating-point code at the instruction level for the following reasons:

- The TOS (top of stack) value of the floating-point status word is set to 0 after each MMX™ instruction. This means that the floating-point code loses its pointer to its floating-point registers if the code mixes MMX instructions within a floating-point routine.
- An MMX instruction write to an MMX register writes ones (11s) to the exponent part of the corresponding floating-point register.
- Floating-point code that uses register contents that were generated by the MMX instructions may cause floating-point exceptions or incorrect results. These floating-point exceptions are related to undefined floating-point values and floating-point stack usage.
- All MMX instructions (except EMMS) set the entire tag word to the valid state (00s in all tag fields) without preserving the previous floating-point state.
- Frequent transitions between the MMX and floating-point instructions may result in significant performance degradation in some implementations.

If the application contains floating-point and MMX instructions, follow these guidelines:

- Partition the MMX™ technology module and the floating-point module into separate instruction streams (separate loops or subroutines) so that they contain only instructions of one type.
- Do not rely on register contents across transitions.
- When the MMX state is not required, empty the MMX state using the EMMS instruction.
- Exit the floating-point code section with an empty stack.

Example 32-2. Floating-point (FP) and MMX™ Code

```
FP_code:
    ..
    ..  (*leave the FPU stack empty*)
MMX_code:
    ..
    EMMS (*mark the FPU tag word as empty*)

FP_code 1:
    ..
    ..  (*leave the FPU stack empty*)
```

32.5.5 Using MMX™ Code in a Multitasking Operating System Environment

An application needs to identify the nature of the multitasking operating system on which it runs. Each task retains its own state which must be saved when a task switch occurs. The processor state (context) consists of the general-purpose registers and the floating-point and MMX registers.

Operating systems can be classified into two types:

- Cooperative multitasking operating system.
- Preemptive multitasking operating system.

The behavior of the two operating-system types in context switching is described in "Context Switching" in Chapter 10 of the *Intel Architecture Software Developer's Manual, Volume 3*.

32.5.5.1 COOPERATIVE MULTITASKING OPERATING SYSTEM

Cooperative multitasking operating systems do not save the FPU or MMX state when performing a context switch. Therefore, the application needs to save the relevant state before relinquishing direct or indirect control to the operating system.

32.5.5.2 PREEMPTIVE MULTITASKING OPERATING SYSTEM

Preemptive multitasking operating systems are responsible for saving and restoring the FPU and MMX state when performing a context switch. Therefore, the application does not have to save or restore the FPU and MMX state.

32.5.6 Exception Handling in MMX™ Code

MMX instructions generate the same type of memory-access exceptions as other Intel Architecture instructions. Some examples are: page fault, segment not present, and limit violations. Existing exception handlers can handle these types of exceptions. They do not have to be modified.

Unless there is a pending floating-point exception, MMX instructions do not generate numeric exceptions. Therefore, there is no need to modify existing exception handlers or add new ones.

If a floating-point exception is pending, the subsequent MMX instruction generates a numeric error exception (interrupt 16 and/or FERR#). The MMX instruction resumes execution upon return from the exception handler.

32.5.7 Register Mapping

The MMX registers and their tags are mapped to physical locations of the floating-point registers and their tags. Register aliasing and mapping is described in more detail in Chapter 10, *MMX™ Technology System Programming Model*, in the *Intel Architecture Software Developer's Manual, Volume 3*.